Engineering Science and Technology, an International Journal 17 (2014) 218–226

HOSTED BY

Contents lists available at ScienceDirect

Engineering Science and Technology, an International Journal

journal homepage: http://www.elsevier.com/locate/jestch

Full length article

Generating combinatorial test cases using Simplified Swarm Optimization (SSO) algorithm for automated GUI functional testing

Bestoun S. Ahmed*, Mouayad A. Sahib, Moayad Y. Potrus

Software Engineering Research Group, Software Engineering Department, Engineering College, Salahaddin University-Hawler, Erbil, Iraq

ARTICLE INFO

Article history:
Received 26 April 2014
Received in revised form
7 June 2014
Accepted 26 June 2014
Available online 6 August 2014

Keywords:
GUI functional testing
Model-based testing
Combinatorial testing (CT)
Simplified particle swarm optimization
Search base software engineering (SBSE)

ABSTRACT

Graphical User Interface (GUI) is the outer skin of programs that facilitate the interaction between the user and different type of computing devices. It is been used in different aspects ranging from normal computers, mobile device, to even very small device nowadays like watches. This interaction uses different tools and programming objects like images, text, buttons, checkboxes, etc. With this emergence of different types of GUIs, they become an essential component to be tested (if available in the software) to ensure that the software meets the required quality by the user. In contrast to non-functional testing, function testing of GUI insures a proper interaction between the user and the application interface without dealing with the coding internals. In this paper, a strategy for GUI functional testing using Simplified Swarm Optimization (SSO) is proposed. The SSO is used to generate an optimized test suite with the help of Event-Interaction Graph (EIG). The proposed strategy also manages and repairs the test suites by deleting the unnecessary event sequences that are not applicable. The proposed generation algorithm based on SSO has proved its effectiveness by evaluating it against other algorithms. In addition, the strategy is applied on a standard case study and proved its applicability in reality.

1. Introduction

Nowadays, computers play an important role in our daily lives both economically and socially. As an effective key issue in this social and economical development, software systems are becoming more challenging to build. With this emergence, it is no longer acceptable to the society to develop poor quality software since it might causes loss of data, life, or fortune. For this reason, testing represents an important and valuable phase in the software development life cycle.

The main aim of the testing phase is to identify all defects existing in the software under test [1]. However, the defects may come from different sources depending on the application and the software design. For this reason, many software testing methods have been developed, such as Equivalence Class Partitioning, Boundary Value analysis, and Cause – Effect Graphing. Each method is used to find a specific type of defect. Hence, as a rule of thumb, software testers always apply more than one method for testing to make sure that the program has a good quality to release.

As a program interface, Graphical User Interface (GUI) represents the internal functionality of the actually program code by visual components, i.e., standard formats for representing text and graphics. Functional testing of a GUI deals with how the keyboard and mouse inputs are handled by the application and whether the application under test performs properly with images, buttons, menus, text, … etc [2]. This testing process is held without dealing with the internal structure of the program but it just deals with the functionality of it. Recently, evidence showed that in software systems, one of the common causes of defects comes from the combination among the individual component of GUI by catching the event transactions [3]. Normally, these kinds of defects cannot be captured by the ordinary testing methods. However, the events could be represented more formally with the help of event-based modeling [4]. From this point, the GUI combination testing methods have been appeared to overcome this problem by considering the combinations among the events [5].

Generally, the combination testing (also known as combinatorial testing) methods depends mainly on a well-known mathematical theory called combinatorial design. The combinatorial design concerns with the arrangement of finite set of elements into patterns (subsets, words, arrays) according to specified rules [6]. As a part of this theory, the arrangement of combinations could be represented in the Covering Array (CA) form which represents all

* Corresponding author. Tel.: +964 750 1725998.
E-mail addresses: bestoon82@gmail.com (B.S. Ahmed), sahib@eng-usalah.org (M.A. Sahib), moayad_75@yahoo.com (M.Y. Potrus).
Peer review under responsibility of Karabuk University.

http://dx.doi.org/10.1016/j.jestch.2014.06.001

the combinations in one array. Thus, combinations number can be reduced dramatically but all of them are covered by the CA. Evidence showed that this reduction can be very effective in term of defect detection since there is no need to test all possible combinations [7].

Generating an optimized list of combinations for a specific input is a painstakingly difficult task because it is an NP-hard computational optimization problem [7]. There is an exponential increase in the computational time as well as in the degree of problem complexity with the increase of the input parameters. For this reason, there is a need for an efficient and intelligent strategy for generation in order to get the most optimum set of combinations with an affordable time. From the literature, there are different tools and strategies for generating the combinatorial set of different inputs [7]. Each tool uses a special algorithm for generation. Among those implemented algorithms, biology inspired algorithms have been implemented recently to overcome this problem. Based on the experimental studies, the use of these algorithms and theories with the random generation proves the generation of optimal combination sizes [7,8]. Many strategies including the stochastic population-based algorithms have been developed. Recent researches demonstrate that strategies based on Ant Colony Algorithm (ACA) [9], Simulated Annealing (SA) [10], Genetic Algorithm (GA) [11], Tabu Search (TS) [12], and Particle Swarm Optimization (PSO) [13–15] can effectively generate optimized test suites in term of size.

Previous studies showed the effectiveness of PSO to generate test suites for different experimental tests due to its robustness and simplicity [14,16]. However, the problem with the conventional PSO is that the convergence speed decreases as the number of iteration is increased, which affects the particles to achieve the best value [17]. To overcome this drawback, Simplified Swarm Optimization (SSO) algorithm has been introduced in the literature [18,19]. Due to such appealing prospects whilst complementing earlier works on GUI testing, this research uses SSO algorithm to optimize the size of the GUI testing suite.

With this evolving scene, the contributions of the research are threefold. Firstly, SSO is applied to the generation of combinatorial test suits to overcome the drawback of the conventional PSO. The SSO helps to generate better results in terms of generation size and time. Secondly, unlike other related researches in this direction, functional testing techniques are used to consider not only the click events of the GUI, but to consider other events like keyboard inputs. The third contribution of the research is the development of a new strategy that contains different stages of generations including the generation of combinations, optimizing the combinations using SSO, and repairing the generated test suites to remove unwanted test cases.

The rest of the paper is organized as follows. Section 2 presents the theoretical background, mathematical notations, and definitions of the combinatorial optimization. Section 3 illustrates in detail the event-based modeling and identifies the problem of the research using a well-know practical example. Section 4 illustrates how the combinatorial optimization utilized with the event based modeling and GUI testing. Section 5 summarized recent related works. Section 6 gives an extensive review for SSO. Section 7 discusses the design and implementation of the proposed strategy, including its structure. Section 8 illustrates how the SSO algorithm is used within the strategy to optimize the generated combinatorial test suite. The evaluation results are presented in Section 9. Section 10 concludes the paper.

2. Mathematical representation of combinatorial optimization

Generally, combinatorial design searches for best solution from finite set of feasible solutions. It is essential to cover all combinations at least once for any combinatorial set. Mathematically, Covering Array (CA) has been introduced to represent all those combinations. A CA_λ (N; t, k, v) represents an ($N \times k$) array with v values such that every ($N \times t$) sub-array contains all ordered subsets from v values of size t at least λ times [20] where k is the number of components (parameters). For optimal combination-set, it is desired that all t-combinations occur at least once. Hence, (t) represents combination degree. In this case, we consider the value of ($\lambda = 1$), and the notation becomes CA (N; t, k, v) [21]. As we are searching for optimal set, the size of N, which is the size of the combination-set have to be as minimum as possible. Fig. 1 illustrates the mechanism of combination minimization and coverage by CA using a simplified example.

As shown in Fig. 1, the covering array CA (4; 2, 2^3) represents an array of size ($N = 4$) and three parameters each of them has two values for representing the interaction of two parameters. As illustrated by the colors, each row in the CA can cover three interactions. Hence, with a CA of size four (i.e. $N = 4$) we can cover 12 interactions. In other words, the covering array CA (4; 2, 2^3) has the potential of 12 interactions at once. Another variation of CA is Mixed Covering Array MCA (N; t, v^p) [22] which is used when the parameters values varied for mixed level covering array.

3. Event-based modeling definition in GUI using a practical example

Among the many ways of interacting with software, practically GUIs is the most popular way that users interact with any software. This interaction is done icons, menus, and windows that can be manipulated by mouse. GUIs have gained more importance recently because of the different forms that have been developed especially those used for mobile applications.

With the emergence of GUI, most of its testing techniques were incomplete and used ad hoc testing manually. However, with the recent development in software techniques and tools, there appears a need to formalize these techniques and also to automate them. To this end, researches focus mainly on three formalizations uses: (1) finite state machines [23], (2) pre and post-conditions [24], and (3) directed graph models [25]. Among those techniques the use of direct graph models appears to be the most impressive technique since it shows promising results in the literature.

Often, in the graph model we try to catch all the executed sequence of the GUI under test by modeling all possible events using the event–flow graph (EFG) [26]. The produced model must contain the events and the relationship among them. The events are represented by node (N) and the relationship by (edge). As an example, consider the simplified GUI shown in Fig. 2.

Considering the "*File*" menu in Fig. 2(a), clearly there are five events which are: *File*, *New*, *Open* … , *Save*, and *Save As* …. Fig. 2(b) shows the nodes and edges when using EFG representation form. Generally, consider the two nodes denoted by n_x and n_y. An edge from n_x to n_y represent the path of execution between n_x to n_y when n_y event is executed immediately after the n_x event. To facilitate the representation, the directed edges in the EFG are represented by a set (E) of ordered pairs (e_x, e_y), where $\{e_x, e_y\} \subseteq N$ and $(e_x, e_y) \in E$ if e_y follows e_x. Although this representation gives a useful base of the testing process, the events that open or close menus, or open windows are considered which are not actually a testing event rather than they are event to start the actual events for testing. To overcome this issue, a refinement GUI modeling has been recently developed called event–interaction graph (EIG) [27]. The EIG is a variant from EFG such that it can achieve more compact and efficient model. In such a model, the menu-opening "*File*" event for the GUI is neglected. Hence to form EIG, the EFG

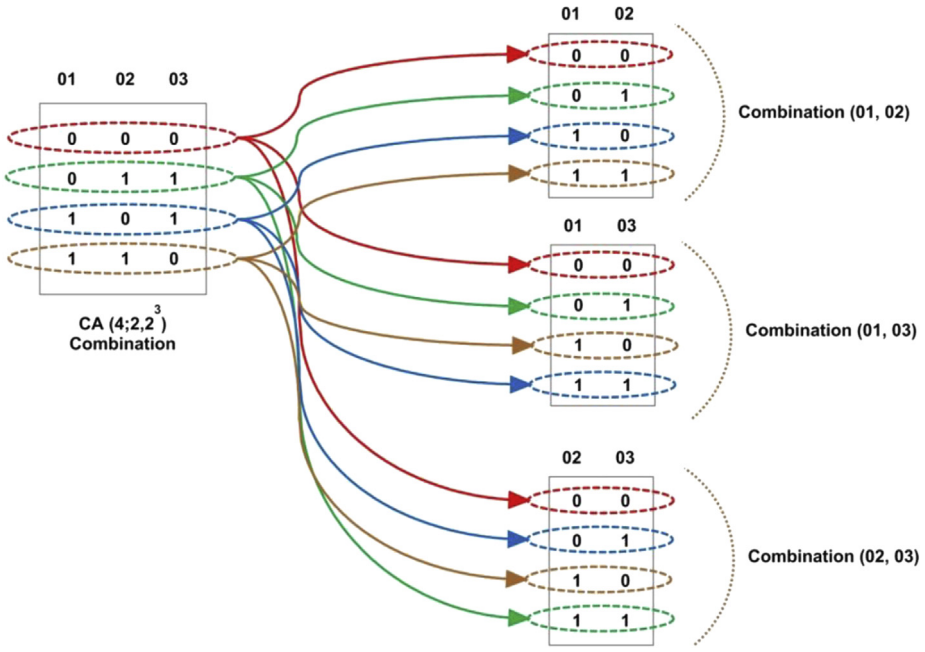

Fig. 1. An illustration of the combination coverage by the CA.

in Fig. 1(b) is reconstructed by neglecting the "File" event then for all remaining events, the even after "File" is replaced by the "File" in the edge. Hence, each (e_x,*File*) edge is replaced with the (e_x, e_y) edge for each occurrence of (*File*, e_y) edge; also for e_y, all edges (*File*, e_y) have to be deleted. Fig. 3 illustrates the EIG graph generated from EFG.

EIG often used to derive the test cases for all model edges. Smoke testing technique has been used to derive the test cases for EIG [25]. Each test in this technique is representing an edge. Hence from Fig. 3, three smoke test cases are, <New, Open>, <New, Save> and <New, Save As> out of 14 test cases, one for each edge. From this point we can observe that the events open or close the menus (i.e., "File" this case) are not considered as test cases. However, in the execution time, they will be considered with a mapping concept {New → (File, New), Open → (File, Open), Save → (File, Save)} as recommended by [25]. Thus, the new test cases will be in the form <File, New, File, Open>, <File, New, File, Save> and <File, New, File, Save As>

4. Utilization of combinatorial design

Recently the combinatorial design and CA notations have been adopted with EIG to derive the test cases. The use of this technique has proved its effectiveness in detecting faults that were previously undetected by other techniques [3]. The reason behind using this method is that we cannot test everything when we choose to test a GUI with many configuration or many events. The combinatorial design will reduce the number of test cases also will check the faults caused by the interaction among the events by using a systematic sampling method depending on the combinations among the events.

For example, if we have five locations in the GUI each of them has three events; we need 3^5 or 243 test cases to test the GUI exhaustively. However, using the CAs properties, we can systematically sample those events. Using the combination of two events, for example, we can test by only 11 test cases and the notation will be CA (11; 2,5,3).

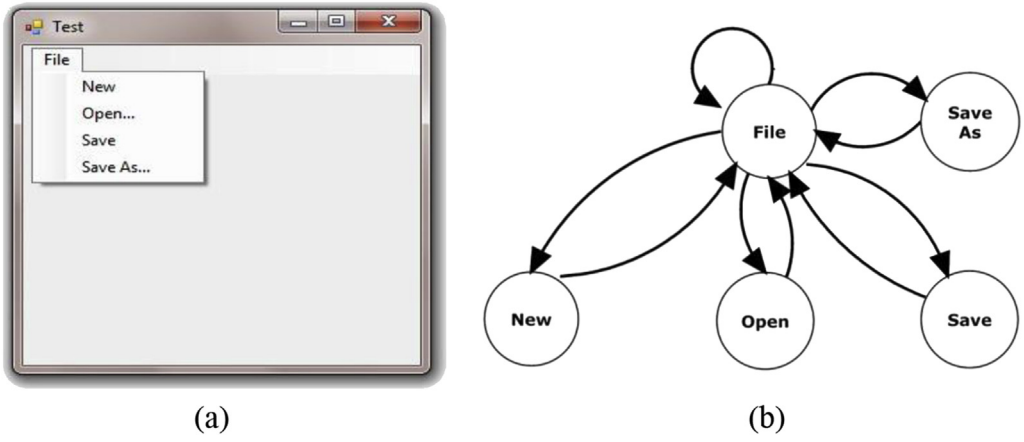

Fig. 2. Notepad application example (a) GUI (b) EFG.

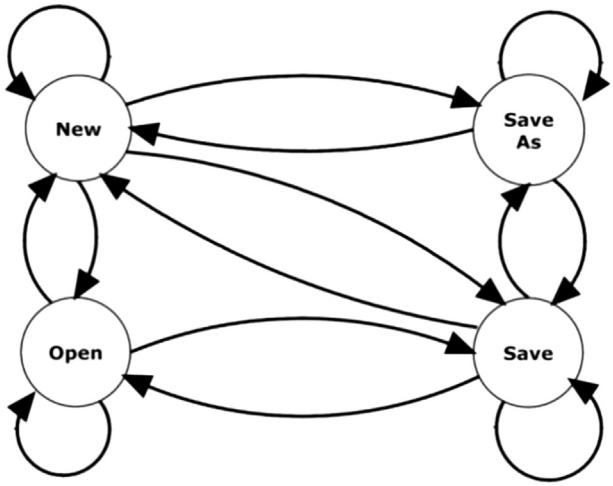

Fig. 3. EIG graph for the GUI of Fig. 2.

When the CA is generated, there would be some incomplete or incorrect events which leads to introduce some constraint [3]. For example, in order to open some event, there is a need to execute some other event first or there are some events that could not be executed in the same time. As in Fig. 2 for example, we cannot run the "Save" event without running the "Save As … " event first.

5. Related work

Several research attempts have been made to address the test cases generation matter and how to solve the constraint problem. A new feedback-based technique for GUI testing has been developed by Yuan et al. [28]. The technique depends on creating and executing an initial seed test suite for the software under test. The seed test suite is generated using the EIG GUI model, then executed using automatic test case re-player method. The test suit is supplemented by a feedback during execution by generating additional test cases. The relationship between event pairs is identified to capture how they are related to each other [28]. The empirical study reveals to the fact that with feedback-based technique there are still infeasible test cases which cannot be run in the test suite.

In another research, Memon tried to address this problem by repairing the unusable test cases [29]. The work was based on determining the usable and unusable test cases automatically from the test suite. Later, the unusable test cases that can be repaired are determined so that they can be executed. The repairing transformations are used to repair the test cases. The study presents useful and effective results, however, there are still many types of constraints that should be solved and dealt with. A method has been developed by Huang et al. to automatically repair GUI test suites and generate new test cases that are feasible [30]. The test suite's coverage is increased by evolving new test cases using genetic algorithm which produces effective results for different types of constraints. The research has showed that the genetic algorithm outperforms the random algorithm in achieving the same goal for almost all cases.

Recently Yuan et al. [3] attempt to define a new criteria for GUI testing using the aforementioned researches grounded in CAs in more detail. The research incorporates a "context" into the criteria in terms of event combinations, sequence length, and by including all possible positions for each event. The criteria are based on both the efficiency (measured by the size of the test suite) and the effectiveness (the ability of the test suites to detect faults). The study conducts more empirical studies using eight GUI applications. Compared to earlier techniques, the results of those studies showed that by increasing the combination degree, and controlling the relative positions of events, large number of faults can be detected.

6. Simplified Swarm Optimization (SSO)

Particle swarm optimization (PSO) is considered to be one of the most promising optimization techniques due to its simplicity, robustness, fast convergence, and ease of implementation. PSO algorithm employs the concept of social interaction for optimization problem solving [31]. It performs an intelligent searching through a defined problem space using a population of individual solutions called particles [32]. The particles are grouped in a finite set of size N called swarm and are updated iteratively [33]. They cooperate by exchanging information with neighbors about what they have discovered in their past search results. In each iteration, a particle has to move to a new position using its past position plus a new updated velocity values [34]. In PSO, updating the velocity for each particle is the most important step of the algorithm. In this step, the velocity is updated using three components; the previous velocity (inertia), a personal influence (cognitive), and social influence (social). The inertia component makes the particle move in the same previous direction and velocity. The cognitive component improves the new particle's position by forcing it to move towards a past position better than the current. The social component makes the particle follow the best neighbor's direction. The modified velocity and position of each particle can be calculated according to the following equations [35]:

$$V_i^{(t+1)} = wV_i^{(t)} + c_1 r_1 \left(P_i - X_i^{(t)}\right) + c_2 r_2 \left(P_g - X_i^{(t)}\right) \quad (1)$$

$$X_i^{(t+1)} = X_i^{(t)} + V_i^{(t+1)} \quad (2)$$

where $X_i^{(t)}$ and $V_i^{(t)}$ are the i-th swarm particle represented by a D dimensional position vector ($X_i^{(t)} = [x_{i1}, x_{i2}, \ldots, x_{iD}]$) and the i-th swarm particle's velocity vector ($V_i^{(t)} = [v_{i1}, v_{i2}, \ldots, v_{iD}]$) both defined at iteration t respectively. $P_i = [p_{i1}, p_{i2}, \ldots, p_{iD}]$ is the i-th swarm best particle's position that has been visited so far. $P_g = [p_{g1}, p_{g2}, \ldots, p_{gD}]$ is the best particle position among all the swarm particles. The variables w, (c_1, c_2), and (c_1, c_2) are the inertia weight factor, acceleration constants, and random numbers between 0 and 1 respectively [18].

Practically, the search space of PSO algorithm is bounded according to the visible solutions of the application. In addition, it is also important to impose limitations on the distance of the particle's movement. This is done by clamping the particle's absolute velocity to a maximum velocity limit [36].

Various studies exist on how to develop the PSO algorithm to perform better and increase its application to complex optimization problems by modifying Equation (1). The attempts of past research were to make the PSO method more complex, as this can achieve increase of the algorithm's adaptability to other optimization problems. This study takes the opposite approach and simplifies the PSO method.

The simplified version of PSO (SSO) is implemented by eliminating the personal influence (cognitive) term in the velocity modification equation. This is accomplished by setting $c_1 = 0$ in Equation (1) thus it becomes:

$$V_i^{(t+1)} = wV_i^{(t)} + c_2 r_2 \left(P_g - X_i^{(t)}\right) \quad (3)$$

The steps involved in SSO algorithm are as follows:

(a) Initialize randomly the positions X_i and velocities V_i of each particle in the swarm.(The initial values must fall within the boundaries defined to the positions and velocities)

(b) Calculate the fitness value of each particle in the swarm using the problem's objective function.
(c) Based on the results of step (b) modify the best particle position among all the swarm particles P_g.
(d) Update the particle's velocity and position vectors using Equations (3) and (2) respectively.
(e) Repeat steps (b), (c), and (d) iteratively until the best fitness value or the maximum generation is met.

The SSO is found to give somewhat improved performance and also makes it easier to tune the behavioral parameters. The simplified PSO is also called Many Optimizing Liaisons (MOL) to make it easy to distinguish from the original PSO. MOL differs from PSO in that it eliminates the particle's best known position thus making the algorithm simpler.

7. The proposed testing strategy

Fig. 4 shows an abstract block diagram which summarizes the steps of the proposed strategy while Figs. 5 and 6 show the main interfaces of the strategy. The strategy starts by receiving the input locations and their events. Then, the combination degree for each test is specified to assign the event combination for the GUI. Based on the specified combination degree, the strategy generates the GUI combination list that contains all the events (i.e., all combinations list). However, the generated list contains all event combinations including the combinations that are redundant. Hence, the generated list is repaired by deleting the redundant (unnecessary) events by providing the constraint events.

Each round of the SSO algorithm, a test case is generated that cover most of the event combination. The output of the proposed strategy is a test suite that contains all the combinations of the GUI events in an optimized form. Section 8 illustrates the detail process of the optimization algorithm where SSO is used.

8. The use of SSO within the strategy

In this research, the SSO algorithm starts when the "combination list" is generated (See Fig. 4). The SSO algorithm uses the generated list as a base to form the fitness function. This fitness function is used in the current algorithm to find the best row in the CA in which it covers most of the combinations in the combination list (i.e., larger number of rows in the list). At the initialization stage

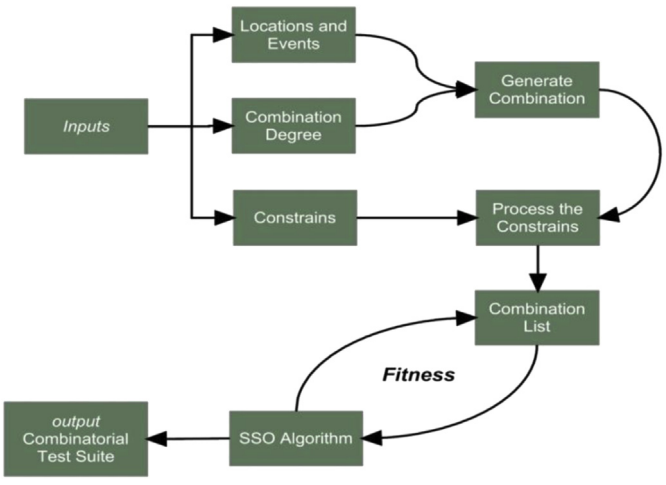

Fig. 4. An abstract representation of the necessary steps accomplished by the implemented strategy.

of the algorithm, random swarm is generated with a discrete start and end of the values for each component. Each particle in the swarm search space takes the form of a vector with elements equals to the number of input locations. Each element contains the events for its corresponding components. In the same time, for each particle in the swarm, there will be a row of velocity represented by a vector and corresponds to each element of the particle. The velocity vector is filled randomly. After the initialization, the search space is updated base on Equations (2) and (3).

In the update process, there is a possibility to produce non-integer values of the velocity that leads to produce out of range values in the search space. To avoid this situation, whenever this situation appears there is a condition to round the velocity to the nearest integer number. In addition, a constraint condition is provided to the values to avoid them going out of range by restricting the values of the velocity to both lower and higher bounds. The boundary condition is set in such a way when the velocity reaches a certain dimension bound; it continues its motion with the same velocity, starting from the other bound of that dimension.

By setting these constraints, for each iteration, the algorithm takes all the rows in the search space and measures the coverage for each of them one by one. The coverage is measured by the number of combination that it can cover. At the end of iterations, the best coverage row is chosen as a best test case and the combination that it covers will be deleted. Then the search space is updated and the algorithm iterates again to search for test cases. This process will continue until the combination list will get empty (i.e., all combination will be cover).

It is worth mentioning that the choice of the best design parameters for the SSO algorithm depends on the application and needs more study. For the purpose of this research the design parameter of the algorithm is set carefully to achieve best results most of the time depending on our experience with PSO and adopting from [18]. The number of particles and iteration were chosen base on the trails that achieve best results. As a result, the number of particles is set to 40 and the number of iterations to 100. In addition, w is set to 0.5 and c to 1.

9. Evaluation

In this section the proposed testing strategy is evaluated. For a systematic evaluation process, two stages, considering the efficiency and effectiveness, are covered: (1) comparing with other strategies in terms of generated time and size to know the efficiency of the proposed strategy; (2) make an empirical study to know the effectiveness by applying the strategy on a real world case study. The following sub-sections summarize these stages.

9.1. Comparing with others strategies

Here, the proposed testing strategy is evaluated by comparing it with other well-known strategies. These strategies include: Pairwise Independent Combinatorial Testing (PICT) [37], Test Vector Generator (TVG) [38,39], Classification-Tree Editor eXtended Logics (CTE-XL) [40], Intelligent Test Case Handler (ITCH) [41], and In Parameter Order Generator (IPOG) [42]. In addition, the SSO strategy is compared with the conventional PSO strategy [14]. To insure a fair comparison, all strategies run within an environment consisting of a desktop PC with Windows 7, 2.8 GHz Core 2 Duo CPU, 2 GB of RAM, and C# installed.

Due to its dependency on some degree of randomness, SSO produce non-deterministic results. Base on the literature, the results for the non-deterministic strategies achieved by running each experiment 5 or 10 times then best sizes is chosen accordingly [9,43–45]. For the case of this research, each experiment is

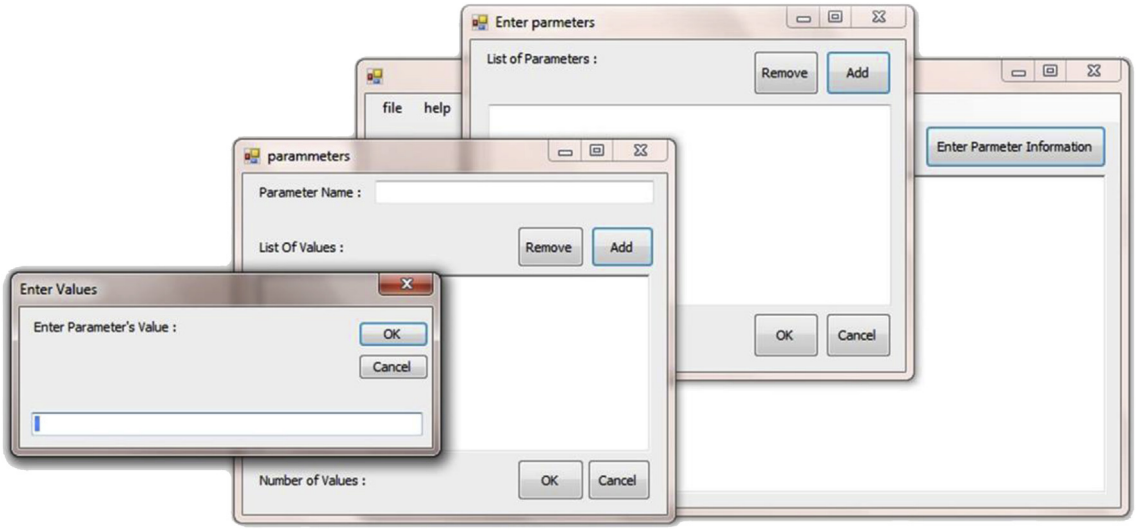

Fig. 5. Main Interface of the Proposed Strategy.

performed 50 times in accord its statistical significance. In addition, for better demonstration of the performance, the best generated size (Bst. Size) and the average generated size (Avg. Size) is reported for each experiment.

For the purpose of evaluation, three sets of experiments have been used. Each set represent a real GUI configuration with different events. In the first experiment, a GUI is used with exactly seven locations each having three events with variable combination degree varying from two to six. In the second experiment, a GUI is used with different locations (4–10) each having three events and fixed combination degree equal to three. In the third experiment, a GUI is used with exactly seven locations and the events are varied in each of them (2–6) and fixed combination degree equal to three. Tables 1–3 show the experimental results for the first, second, and third experiments respectively.

The results in Tables 1–3 shows the size of the produced test suite and the time in seconds for each one to be generated. To give better indication for both PSO and SSO strategies, best and average sizes (i.e., Bst. Size, and Avg. Size) have been reported since they have randomness in the algorithm. For the other strategies, just the best size is reported because they depend on the designed algorithms (with no randomness) not any artificial intelligence theories. The better sizes are shown in bold numbers while the best generation times are shown in shaded cells.

From Table 1–3, we can observe that when the number of events or the combination degree is increased, the generation time will increase exponentially. However, in one side those strategies using computational algorithms generate the test cases faster than AI algorithms; they are not able to generate better sizes than AI. This is because AI strategies need more iteration to carry out the optimization process. Considering the AI based strategies (i.e., PSO and SSO in this case), SSO strategy can generate results similar or better than PSO but with better generation time. This is because of the simplified structure of SSO compared to PSO.

9.2. Empirical study

In order to evaluate the effectiveness of the proposed strategy, a case study is conducted on a reliable artifact program. Here, the goal is to validate the correctness of the strategy and demonstrate its feasibility and applicability.

In this case study, the use of automated testing software (Quick Test Professional – QTP) is incorporated for the purpose of testing. The tool is used for the purpose of functional testing which is useful for ActiveX controls, Web objects, and Basic Control package written in Visual Basic scripting language. As a case study, the “flight reservation” artifact is considered which is a program written in “Visual Basic” and comes with QTP for functional testing purposes.

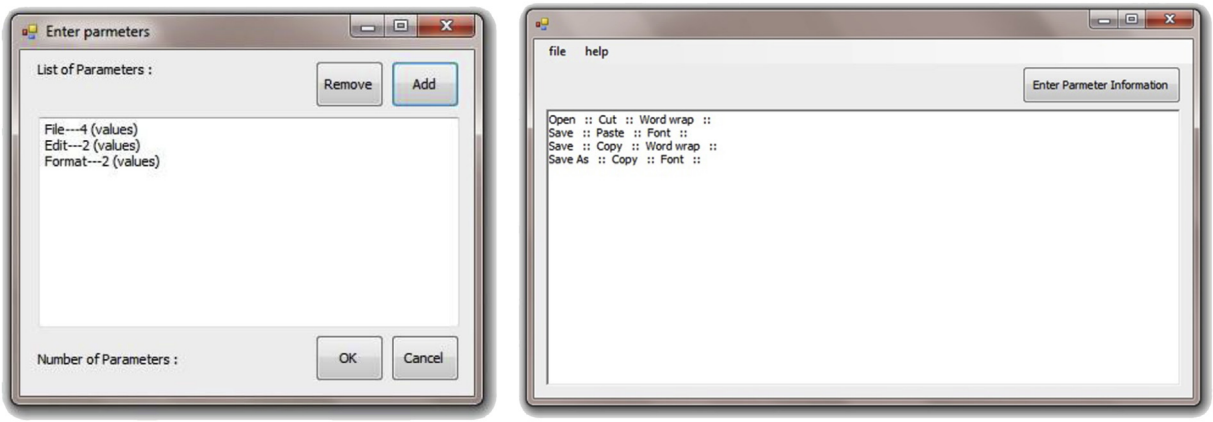

Fig. 6. Locations and events representation in the interface before and after generation.

Table 1
Test sizes and execution times for a GUI with 7 locations, each having 3 events when the combination degree is varied up to 6.

Combination degree	PICT	TVG	CTE-XL	IPOG	PSO		SSO	
	Size/time	Size/time	Size/time	Size/time	Bst. size/time	Avg. size/avg. time	Bst. size/time	Avg. size/avg. time
2	16/0.62	**15**/0.22	16/0.26	17/0.443	**15**/0.21	15.23/0.32	**15**/0.19	15.13/0.29
3	51/0.98	55/0.57	54/2.55	57/0.614	50/4.21	51.75/5.56	**50**/2.23	51.42/3.18
4	168/1.46	167/0.82	NS	185/1.357	155/11.32	157.77/13.95	**153**/10.21	157.47/11.72
5	452/2.27	464/4.602	NS	608/2.264	**441**/41.05	445.96/44.74	**441**/38.11	441.13/40.52
6	1015/3.29	1016/11.524	NS	1281/3.97	**977**/105.59	980.48/109.54	**974**/91.35	980.48/93.51

Table 2
Test sizes and execution times for a GUI with locations (4–10) each having 3 events when the combination degree is 3.

Location	PICT	TVG	CTE-XL	IPOG	PSO		SSO	
	Size/time	Size/time	Size/time	Size/time	Bst. size/time	Avg. size/avg. time	Bst. size/time	Avg. size/avg. time
4	34/0.14	34/0.17	34/0.75	39/0.27	**27**/0.17	29.3/0.32	**27**/0.15	28.5/0.25
5	43/0.45	41/0.21	43/1.44	43/0.34	**39**/1.739	41.37/2.56	**39**/0.95	41.13/1.26
6	48/0.83	49/0.48	52/1.96	53/0.58	**45**/2.25	46.76/3.1	**45**/1.12	46.38/2.43
3	51/0.98	55/0.57	54/2.55	57/0.614	50/4.21	51.75/5.56	**50**/2.23	51.42/3.18
8	59/1.3	60/1.251	63/2.85	63/0.98	**54**/7.15	56.76/9.2	**54**/3.67	56.76/4.11
9	63/2.76	64/1.812	66/4.65	65/1.36	**58**/9.03	60.30/12.8	**58**/6.84	60.30/8.65
10	65/2.94	68/2.414	71/5.9	68/1.92	**62**/13.27	63.95/16.73	**62**/9.69	63.95/10.96

As the name indicates, the application is used to reserve tickets for flights from 10 cities towards 10 other cities. The application is been used widely for functional testing as it is non-trivial, and reliable. Fig. 7 shows the main interface of the application.

The application has different locations and for each of them different events. For instance seven locations are chosen from the main interface of the application to be tested. Table 4 summarizes the locations and the description for each of them.

Using the CAs method of abstraction for the combination arrangement, the locations and their events can be represented as MCA (N; t, 3^3 10^2 4^1 2^1). By knowing those locations and events, they are used to generate the test cases within our strategy. Base on the illustration from Fig. 3, the locations and events are provided in addition to the combination degree that we want to test with and the constraints that we may have. The combination is generated first base on the combination degree. Then, the constraints are taken into consideration in the strategy by deleting them in the combination or simply by adding them to the final test cases. Here the faults are counted when the test case cannot perform a require function and cannot be run on the case study. Table 5 shows the number of test cases in each corresponding MCA that have been run on the case study and the faults counted for each set.

Referring to Table 5, it can be observed that the test sets are effective when applied on the case study since a number of faults have been detected. Regarding the first test set, MCA (N; 2, 3^3 10^2 4^1 2^1), when the combination degree = 2, it can show 21 faults. When the combination degree become higher, i.e., MCA (N; 3, 3^3 10^2 4^1 2^1) when the combination degree = 3, more faults could be detected. In the same way when the combination degree = 4 in MCA (N; 4, 3^3 10^2 4^1 2^1), other faults can be detected. However, it can be observed that when the combination degree becomes higher, only two new faults can be observer because most of the faults have been detected already by the other test sets. Hence, it can be concluded that the generated test sets and used technique are able to detect faults effectively.

9.3. Threats to validity

As in other researches, the empirical case study of this research suffers from threats to validity. This research tries to minimize the threats in the experimental design stage. However, still there are some threats that need to be addresses here. Firstly, this research takes only one GUI as a case study. Although this GUI case contains different GUI components which are useful for the subject of GUI functional testing, there could be other examples of GUIs containing other components that may need different testing technique. Secondly, the faults considered in this research are the wrong sequence or order when the test cannot be run on QTP. However, there may be other kind of faults that could not be catch by this technique. Thirdly, with respect to the construction validity, within the current strategy, the test cases are generated and then it is applied on the case study using QTP. This is because QTP does not have the facility of connecting with C#. There could be an equivalent tool for functional testing that has a facility to connect with the generation tool. In this way, the automation testing will be fully automated from the generation to its application.

Table 3
Test sizes and execution times for a GUI with 7 locations, the events are varied in each of them (2–6) when the combination degree is 3.

Events	PICT	TVG	CTE-XL	IPOG	PSO		SSO	
	Size/time	Size/time	Size/time	Size/time	Bst. size/time	Avg. size/avg. time	Bst. size/time	Avg. size/avg. time
2	15/0.37	15/0.25	15/0.32	19/0.93	**13**/0.32	13.61/0.52	**13**/0.29	13.61/0.32
3	51/0.98	55/0.57	54/2.55	57/0.614	**50**/4.21	51.75/5.56	**50**/2.23	51.42/2.42
4	124/1.06	134/0.95	136/5.7	208/0.97	116/21.34	118.13/25.64	116/18.52	118.13/20.14
5	241/1.9	260/2.15	267/20.5	275/2.175	**225**/35.6	227.21/37.73	**225**/27.39	227.21/32.43
6	413/3.74	464/4.458	467/55.6	455/3.514	425/183.56	428/197.43	425/151.21	428/167.19

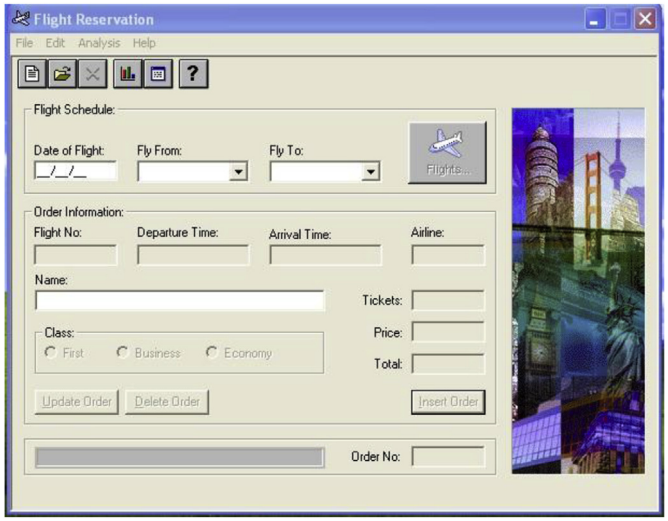

Fig. 7. The main interface of the flight reservation application.

Table 4
Summarization of the location in flight reservation application.

No.	Locations	Description
1	City from	Contains the name of the departure city
2	City to	Contains the name of the arrival city
3	Day	Contains the day of the flight
4	Class name	Contains the class name
5	File	Contain application options
6	Edit	Contain application edit options
7	Analysis	The contain reports and information

Table 5
The size of the used test set and its corresponding detected faults during running.

Test set	Size of test set	No. of counted faults
MCA (N; 2, 3^3 10^2 4^1 2^1)	100	21
MCA (N; 3, 3^3 10^2 4^1 2^1)	416	43
MCA (N; 4, 3^3 10^2 4^1 2^1)	1432	56
MCA (N; 5, 3^3 10^2 4^1 2^1)	4165	58

10. Conclusion

In this paper we have presented a strategy to be used for GUI functional testing. The strategy generates the required test cases for the GUI under test using the combinatorial design and then it removers the unwanted test cases. The Simplified Swarm Optimization theory is used to optimize the test cases considering the combinatorial design concepts. By generating the test cases the tests applied to a real world case study to test the effectiveness of the strategy. The key insight in this work is to leverage the fact that combinatorial design could be used efficiently and effectively for different testing case studies and it could be used also in GUI testing. The strategy has proved its efficiency by generating small test cases in term of size while the time of generating mostly is better than that generated by original PSO version. The strategy also proved its effectiveness by using it within a real world case study and it could catch faults in the application under test.

Acknowledgment

This research is partially supported by the IT center of Engineering College, Salahaddin University – Hawler (SUH). The author would like to thank "MindScripts Technologies", Pune branch, India, for providing the tools and environment for testing also for their help during the training and valuable advices.

References

[1] B.B. Agarwal, S.P. Tayal, M. Gupta, Software Engineering and Testing, Infinity Science Press, Hingham; Toronto, 2010.
[2] W. Yang, Z. Chen, Z. Gao, Y. Zou, X. Xu, GUI testing assisted by human knowledge: random vs. functional, J. Syst. Software 89 (2014) 76–86.
[3] X. Yuan, M.B. Cohen, A.M. Memon, GUI interaction testing: incorporating event context, IEEE Trans. Softw. Eng. 37 (2011) 559–574.
[4] R.C. Bryce, S. Sampath, A.M. Memon, Developing a single model and test prioritization strategies for event-driven software, IEEE Trans. Softw. Eng. 37 (2011) 48–64.
[5] A.M. Memon, GUI testing: pitfalls and process, Computer 35 (2002) 87–88.
[6] D.R. Stinson, Combinatorial Designs: Constructions and Analysis, Springer-Verlag, United States of America, 2004.
[7] C. Nie, H. Leung, A survey of combinatorial testing,, ACM Comput. Surv. 43 (2011) 1–29.
[8] M. Grindal, J. Offutt, S.F. Andler, Combination testing strategies: a survey, Softw. Test. Verif. Rel. 15 (2005) 167–199.
[9] X. Chen, Q. Gu, A. Li, D. Chen, Variable strength interaction testing with an ant colony system approach, in: 16th Asia-Pacific Software Engineering Conference, IEEE Computer Society, Penang, Malaysia, 2009, pp. 160–167.
[10] C.J. Colbourn, G. Kéri, P.P.R. Soriano, J.-C. Schlage-Puchta, Covering and radius-covering arrays: constructions and classification, Discrete Appl. Math. 158 (2010) 1158–1180.
[11] T. Shiba, T. Tsuchiya, T. Kikuno, Using artificial life techniques to generate test cases for combinatorial testing, in: 28th Annual International Computer Software and Applications Conference, vol. 71, IEEE Computer Society, Hong Kong, 2004, pp. 72–77.
[12] K.J. Nurmela, Upper bounds for covering arrays by tabu search, Discrete Appl. Math. 138 (2004) 143–152.
[13] B.S. Ahmed, K.Z. Zamli, A variable strength interaction test suites generation strategy using particle swarm optimization,, J. Syst. Software 84 (2011) 2171–2185.
[14] B.S. Ahmed, K.Z. Zamli, C.P. Lim, Application of particle swarm optimization to uniform and variable strength covering array construction, Applied Soft Comput. 12 (2012) 1330–1347.
[15] X. Chen, Q. Gu, J. Qi, D. Chen, Applying particle swarm optimization to pairwise testing, in: Proceedings of the 2010 IEEE 34th Annual computer Software and Applications Conference, IEEE Computer Society, 2010, pp. 107–116.
[16] B.S. Ahmed, K.Z. Zamli, PSTG: A t-way Strategy Adopting Particle Swarm Optimization, 2010, pp. 1–5. Kota Kinabalu, Borneo.
[17] H. Liu, A. Abraham, W. Zhang, A fuzzy adaptive turbulent particle swarm optimisation, Int. J. Innov. Comput. Appl. 1 (2007) 39–47.
[18] C. Bae, W.-C. Yeh, N. Wahid, Y.Y. Chung, Y. Liu, A new simplified swarm optimization (SSO) using exchange local search scheme, Int. J. Innov. Comput. Inf. Control 8 (2012) 4391–4406.
[19] Y. Wei-Chang, New parameter-free simplified swarm optimization for artificial neural network training and its application in the prediction of timeSeries, IEEE Trans. Neural Networks Learn. Syst. 24 (2013) 661–665.
[20] C. Yilmaz, M.B. Cohen, A. Porter, Covering arrays for efficient fault characterization in complex configuration spaces, ACM SIGSOFT Softw. Eng. Notes 29 (2004) 45–54.
[21] A. Hartman, L. Raskin, Problems and algorithms for covering arrays, Discrete Math. 284 (2004) 149–156.
[22] C.J. Colbourn, Strength two covering arrays: existence tables and projection, Discrete Math. 308 (2008) 772–786.
[23] B. Robinson, L. White, Testing of user-configurable software systems using firewalls, in: 19th International Symposium on Software Reliability Engineering, IEEE Computer Society, Seattle, Washington, USA, 2008, pp. 177–186.
[24] P. Li, T. Huynh, M. Reformat, J. Miller, A practical approach to testing GUI systems,, Empirical Softw. Eng. 12 (2007) 331–357.
[25] A.M. Memon, Q. Xie, Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software, IEEE Trans. Softw. Eng. 31 (2005) 884–896.
[26] S. Arlt, C. Bertolini, S. Pahl, M. Schäf, Trends in model-based GUI TESTING, in: H. Ali, M. Atif (Eds.), Advances in Computers, Elsevier, 2012, pp. 183–222. Chapter 6.
[27] Q. Xie, A.M. Memon, Using a pilot study to derive a GUI model for automated testing, ACM Trans. Softw. Eng. Methodol. 18 (2008) 1–35.
[28] X. Yuan, A.M. Memon, Using GUI run-time state as feedback to generate test cases, in: 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2007, pp. 396–405.
[29] A.M. Memon, Automatically repairing event sequence-based GUI test suites for regression testing, ACM Trans. Softw. Eng. Methodol. 18 (2008) 1–36.
[30] S. Huang, M.B. Cohen, A.M. Memon, Repairing GUI test suites using a genetic algorithm, in: 3rd IEEE International Conference on Software Testing, Verification and Validation, IEEE Computer Society, Washington, DC, USA, 2010, pp. 245–254.

[31] M.R. AlRashidi, M.E. El-Hawary, A survey of particle swarm optimization applications in electric power systems,, IEEE Trans. Evol. Comput. 13 (2009) 913–918.
[32] J. Kennedy, R. Eberhart, Particle swarm optimization, in: IEEE International Conference Neural Networks, IEEE Computer Society, Perth, Australia, 1995, pp. 1942–1948.
[33] A.H. Ertas, Optimization of fiber-reinforced laminates for a maximum fatigue life by using the particle swarm optimization. Part I, Mech. Compos. Mater. 48 (2013) 705–716.
[34] M. El-Abd, H. Hassan, M. Anis, M.S. Kamel, M. Elmasry, Discrete cooperative particle swarm optimization for FPGA placement, Appl. Soft. Comput. 10 (2010) 284–295.
[35] R.J. Kuo, L.M. Lin, Application of a hybrid of genetic algorithm and particle swarm optimization algorithm for order clustering, Decis. Support Syst. 49 (2010) 451–462.
[36] Y. Shi, R. Eberhart, A modified swarm optimizer, in: IEEE International Conference of Evolutionary Computation, Anchorage, AK, USA, 1998, pp. 125–129.
[37] J. Czerwonka, Pairwise testing in real world: practical extensions to test case generator, in: 24th Pacific Northwest Software Quality Conference, IEEE computer society, Portland, Oregon, USA, 2006, pp. 419–430.
[38] J. Arshem, TVG Download Webpage: http://sourceforge.net/projects/tvg, 2010.
[39] Y.-W. Tung, W.S. Aldiwan, Automating test case generation for the new generation of mission software system, in: IEEE Aerospace Conference, IEEE Computer Society, Big Sky, MT, USA, 2000, pp. 431–437.
[40] Y.T. Yu, S.P. Ng, E.Y.K. Chan, Generating, selecting and prioritizing test cases from specifications with tool support, in: 3rd International Conference on Quality Software, IEEE Computer Society, Dallas, Texas, 2003, pp. 83–90.
[41] A. Hartman, IBM intelligent test case handler, in, IBM Alphaworks (2005).
[42] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, IPOG: a general strategy for t-way software testing, in: 4th Annual IEEE International Conference and Workshops on the Engineering of Computer-based Systems, IEEE Computer Society, Tucson, Arizona, 2007, pp. 549–556.
[43] M.B. Cohen, Designing test suites for software interaction testing, in: Department of Computer Science, University of Auckland, 2004, p. 185.
[44] Z. Wang, B. Xu, C. Nie, Greedy heuristic algorithms to generate variable strength combinatorial test suite, in: 8th International Conference on Quality Software, IEEE Computer Society, Oxford, UK, 2008, pp. 155–160.
[45] B. Garvin, M. Cohen, M. Dwyer, Evaluating improvements to a meta-heuristic search for constrained interaction testing, Empirical Softw. Eng. 16 (2011) 61–102.